LANGUAGE PROCESSORS : AN EXERCISE IN SYSTEMATIC PROGRAM DEVELOPMENT

By

A. K. DEY

COMPUTER SCIENCE PROGRAMME
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

JULY 1980

LANGUAGE PROCESSORS : AN EXERCISE IN SYSTEMATIC PROGRAM DEVELOPMENT

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

By

A. K. DEY

to the
COMPUTER SCIENCE PROGRAMME
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR
JULY 1980

CERTIFICATE

This is to certify that the work entitled, "LANGUAGE PROCESSORS : AN EXERCISE IN SYSTEMATIC PROGRAM DEVELOPMENT" has been carried out by Sri A.K. Dey under my supervision and has not been submitted elsewhere for the award of a degree.

Kesav V. Nori

Kanpur
July 1980

Kesav V. Nori
Assistant Professor
Computer Science Programme
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

ACKNOWLEDGEMENTS

With a deep sense of gratitude I thank Professor K.V. Nori. His help and encouragement during the work of this thesis and in general during my stay at IIT Kanpur, has been without measure. He has given freely of his time and his able guidance has given me hope when I most needed it.

I thank my best fried Sanjiv. He has shared my joy and my sorrows for the past two years. Because of him this brief sojourn has been enjoyable.

The typing has been done within a short time and excellently. I thank Mr. H.K. Nathani for this.

Kanpur
July 1980

– A.K. Dey

CONTENTS

ABSTRACT

The thesis discusses the systematic development of a processor for a bootstrappable dialect of PASCAL. The formal specifications for each phase of the processing performed by the processor is presented and corresponding rules are evolved which convert the specifications for each phase into a program. The program for the processor is then synthesized from these specifications with the help of the rules. The phases completed are the lexical analysis, context-free syntax analysis and the context-free error recovery. The formalism required for specifying the context-sensitive syntax analysis is also included. As opposed to normal methods of proof of programs which often become involved when the programs are large and complex, the developmental effort undertaken here provides straightforward insight into the relation between programs and specifications.

CHAPTER 1

INTRODUCTION

1-1. MOTIVATION FOR THE CHOICE OF A BOOTSTRAPPABLE SUBSET OF PASCAL

We are interested in systematically developing a processor for PASCAL (Jensen and Wirth, 74). The intention is that we should be able to argue about the soundness of the method of development so as to obviate the necessity of a proof that the processor meets its specifications. In this regard, our attitude is that we would like to consider the formal specification of the successive phases of processing performed by the processor and stipulate rules of transformation for each of the respective specifications that convent the corresponding specification to a program. In other words, our program is systematically synthesized from the specification itself. This technique would make redundant the difficult task of proving large programs correct (Manna 74).

Rather than start with the entire language PASCAL as the scope for the processor, we would like to consider a process of stepwise enrichment of features of the language : a processor for the simplest subset of the language could be used to implement the next richer dialect of the language.

The choice of a bootstrappable dialect of the language is greatly determined by the resources available for the task of implementation. We already have a PASCAL compiler on the DEC-System-10. So the dialect we choose must be of educative value, not only for purposes of the development of the processor but also for teaching programming, a purported goal in the design of PASCAL. We hope that the dialect

chos.n is neither too small for the purposes above nor too big for implementation c mini and micro-computer systems in which the constraints of memory size is often severe.

1-2. METHODOLOGY OF DEVELOPMENT AND STRUCTURE OF THE THESIS

We will follow the broad outlines of stepwise enrichment of goals set up by (Ammahn 74). However, as our attempt to justify the resulting code is more stringent, we will differ in technique and detail.

The process is envisaged to have the following phases:

(1) Lexical Analysis
(2) Context free syntax Analysis
(3) Recovery from Content Free Errors
(4) Context Sensitive Analysis
(5) Transformations.

We have been successful in completing the first three phases. The formalisms required to tackle the fourth phase are presented. The subject matter for phase (5) depends on the goals of the processor, the toughest to handle being the transformation effected by an optimizing compiler.

The next chapter of this thesis suggests the Regular Expression (RE) formalism for formal specification of Phase (1). The restriction on RE such that they can be deterministically scanned to perform the job of Lexical Analysis is discussed. Rules that convert such restricted REs to PASCAL programs are considered. Some implementation details are added. The entire specification for the Lexical Analysis component is given in Appendix I.

Chapter 3 discusses the problem of formal specification of context free syntax analysis. Naturally, the formalism of Context Free Grammers (CFG) is used. First, we consider a restricted class of CFGs called LL(1).

We consider the notion of the leftmost derivation of a sentence in a LL(1) language. LL(1) is a deterministic class of CFGs. Now, at this stage, a correspondence between LL(1) grammars and PASCAL programs is proposed such that the execution of the program corresponds to a leftmost derivation of the input sentence. Using this correspondance the entire context free parser is synthesised. The LL(1) grammar for PASCAL-B from which the synthesis was effected is given in Appendix III.

Chapter 4 is connected with the problem of error recovery, 'Panic mode' error recovery strategy is used. This is achieved through sets of synchronizing symbols that are computed from the LL(1) grammar of Appendix III. The enrichment of the program generated by transformations described in Chapter 3 to systematically keep track of the synchronizing symbols during a leftmost derivation.

Chapter 5 introduces a formalism called Extended Attribute Grammars that has promise for formal specification of Context Sensitive aspects of PASCAL and the possibility of integration with transformations in Chapters 3 and 4.

Conslusions and suggestion for further work form the last chapter of this thesis.

CHAPTER 2

LEXICAL ANALYSIS

2-1. FUNCTIONS OF THE LEXICAL ANALYSER

The lexical analyser is the interface between the source program and the syntax analyser. The input to the lexical analyser is the source program which is a stream of characters. The lexical analyser groups these characters into symbols where each symbol can be treated as a single logical entity.

By splitting the lexical analysis and the syntax analysis of the source program, the overall design of the processor is simplified because the structure of lexical symbols can be specified by regular expressions.

2-2. REGULAR EXPRESSIONS (RE)

A RE is defined over a finite set of characters, Σ, called the alphabet. A string is a finite sequence of characters from Σ. Symbols of a programming language are strings over the character set of the language (Aho and Ullman 76)

Definition 2-1

The formation rules for REs are:

1. a null string Λ is a RE
2. for $\forall a \in \Sigma$, a is a RE
3. Concatenation of two REs A and B, written AB, is a RE
4. Alternation of two REs A and B written $A \mid B$ is a RE
5. $A^* = \Lambda \mid AA \mid AAA \ldots$ is a RE
6. only expressions obtained by Rules 1-5 are regular expressions.

We would like to deterministically recognize the REs representing the symbols. However, a one-character look-ahead is used. Because of this requirement, we restricted the set of REs to those which can be deterministically recognized.

Definition 2-2

Start (A) : For a RE A

Start (A) = ($\{ a \mid a \in \Sigma$ is a prefix to strings produced by A$\}$)

The restrictions on the REs for deterministic scanning are:

RE	Restriction
AB	if A produces the null string Λ then the sets start (A) and start (B) must be disjoint.
A | B	the sets start (A) and that (B) must be disjoint.

The REs which satisfy the above restrictions can be deterministically recognized.

Now we can specify the construction rules of P(X) which denotes the program schema which recognizes the RE X (Wirth 73). We assume that the variable ch is assigned the next character scanned and that procedure test (x) verifies the equality ch = x. Start (A) denotes the set of starting characters of the RE A.

X	P(X)
(i) a $\in \Sigma$	test (a); read (ch)
(ii) AB	PA; P(B)
(iii) A|B	if ch $\in$ start (A) then P(A) else if ch = start (B) then P(B)
(iv) A*	while ch $\in$ start (A) do P(A)

We present an example program module generated from the rules:

From Appendix I, rule (6),

identifier ::= letter (letter digit)*

The resulting program module is

```
P(identifier) = if ch = letter then
                begin read (ch);
                  while (ch = letter or ch = digit) do
                     read (ch)
                end
```

2-3. IMPLEMENTATION DETAILS

The function of the lexical analyser is

(a) to remove comments and blanks;

(b) to output to the syntax analyser the value of the next symbol in the output;

(c) to list the data as they are read in.

The procedure GETSYM does the lexical analysis. Three tables need to be constructed. The first, WORD, contains the string of characters forming each reserved word; second, WSYM, contains the scalar value of each reserved word and the third, SSYM, indexed by characters, contains the scalar value corresponding to each character. GETSYM calls procedure NEXTCH which makes the next character in the input stream available in the variable CH.

The interface between the lexical analyser and the syntax analyser is the variable SYM. GETSYM makes the value of the next symbol in the input stream available in SYM whenever the syntax analyser calls GETSYM.

The action of GETSYM depends on whether the first character of the next symbol is a letter, a digit, the symbol '=' or otherwise. In the first case the remaining letters and/or digits of the next symbol is packed into a word and the reserved word table searched. In the last case the table SSYM can be used directly to convert the character into a terminal symbol.

The lexical analyser constructed is given in Appendix II.

CHAPTER 3

CONTEXT-FREE SYNTAX ANALYSIS

3-1. INTRODUCTION

The syntax analysis phase has the following functions:

(a) to check that the symbols appear in the patterns that are permitted by the syntax, and

(b) to make the hierarchical structure of the incoming symbol stream explicit by identifying the symbols that should be grouped together.

The specifications for this phase can be formally given through context-free grammars.

The syntax analyser constructed is a recursive descent syntax analyser. The formalization upon which recursive descent parsers are based follows in the next section.

Definition 3-1

A context-free grammar is $G = (N,T,P,S)$ where N and T are finite sets that represent non-terminal and terminal symbols respectively and N and T do not intersect; S, called the axiom is an element of N; P the set of productions, is a set of pairs (A,v) where A is a nonterminal and $v \in (N \cup T)^*$.

3-2. LL(1) GRAMMARS

The syntax of the language being processed can be specified by a restricted class of CFGs called LL(1).

A CFG is said to be LL(1) if a one-look ahead symbol is always sufficient to choose between productions of the grammar which have the same left-hand side while parsing a sentence in the language defined by the grammar (Backhouse 79).

Now we will present a number of definitions which lead to a formal definition of LL(1) grammars.

Definition 3-2

Let $G = (N,T,P,S)$ be a grammar. A string w' is directly derived from a string w if and only if $w = sut$, $w' = svt$ and $u \rightarrow v$ is a productic of G, where s and t are arbitrary strings.

Definition 3-3

A string w' is derived from w if either $w' = w$ or there is a sequenc of strings $w_0, w_1, \ldots, w_n$ such that $w = w_0$, $w' = w_n$ and w_i directly derives w_{i+1} for each i, $0 \leq i < n$. The sequence

$$w_0 \Rightarrow w_1 \Rightarrow w_3 \Rightarrow \cdots \quad \Rightarrow w_n$$

is called a derivation sequence of length n.

Definition 3-4

Let $G = (N,T,P,S)$ be a CFG. The functions NULLABLE, FIRST and FOLLOW are defined on $N \cup T$ as follows:

$$\begin{aligned} \mathrm{NULLABLE}(X) &= \text{true if } X \Rightarrow^* \Lambda \\ &= \text{false otherwise} \end{aligned}$$

$$\mathrm{FIRST}(X) = \{ t \mid t \in T \text{ and } X \Rightarrow^* tw \text{ for some } w \in T^* \}$$

$$\mathrm{FOLLOW}(X) = \{ t \mid t \in T \text{ and } S \Rightarrow uXtw \text{ for some } u \in T^* \text{ and } w \in T^* \}$$

Definition 3-5

The function LOOKAHEAD is defined on the production of G by

$$\begin{aligned} \mathrm{LOOKAHEAD}\ (A \rightarrow X_1 X_2 \ldots X_m) \\ = \bigcup_{i \text{ st. } P_i} \mathrm{FIRST}(X_i) \cup (\underline{\text{if}}\ \mathrm{NULLABLE}\ (X_1 \ldots X_m)\ \underline{\text{then}}\ \mathrm{FOLLOW}\ (A) \text{ else } \emptyset) \end{aligned}$$

where P_i denotes $1 \leq i \leq m$ and NULLABLE $(X_1 \ldots X_{i-1})$

Definition 3-6

The grammar is strong LL(1) if and only if for each pair of distinct production $A \rightarrow \alpha$ and $A \rightarrow \beta$, say, having the same left-hand side,

$$\mathrm{LOOKAHEAD}(A \rightarrow \alpha) \cap \mathrm{LOOKAHEAD}(A \rightarrow \beta) = \emptyset$$

The grammar given in Appendix III is strong LL(1). Now we show the correspondence between an LL(1) grammar and a recursive descent parser.

3-3. CORRESPONDANCE BETWEEN AN LL(1) GRAMMAR AND A RECURSIVE DESCENT PARSER

Consider an LL(1) grammar $G = (N,T,P,S)$. Construct a recursive descent parser program such that for every $A \in N$ a procedure pA is declare and for every $a \in T$, a is scanned by the lexical analyser.

The body of pA is constructed from all $p \in P$ such that p is of the form (A,v) where $v = (N \cup T)^*$.

The rules for constructing the procedure body follows.

v	Program construct
1. AB where $A,B \in N$	p(A); p(B)
2. A | B	if (symbol = FIRST(A)) then p(A) else p(B).
3. $(A)^*$	while (symbol = FIRST(A)) do pA.

Now we shall show the relationship between RD parsers and a left-most derivation sequence.

3-4. LEFT-MOST DERIVATION SEQUENCE AND THE ACTIONS OF A RD PARSER

If $a_0 \Rightarrow a_1 \Rightarrow a_2 \ldots \Rightarrow a_m$ is a left-most derivation sequence, then we know that a_{i+1} is obtained from a_i by applying a production to the leftmost nonterminal in a_i for all i, $0 \leq i < m-1$.

A recursive descent parser, the procedure for the axiom of the grammar starts out by inspecting the current input and selects a productic

to be applied. Since the language is LL(1) this choice is deterministic and is equivalent to transferring control to a procedure for the nonterminal appearing at the leftmost position of the rhs of the production. When the current input symbol matches the leftmost symbol of the rhs while applying a production, the input is advanced. At any point during compilation the state of the parser is represented in the symbols scanned so far concatenated with the parser's continuations. The state of the parser represents a leftmost derivation and the sequence of actions of a recursive descent parser corresponds to a leftmost derivation sequence.

Based upon the theory of this section and Section 3-3, the syntax analyser can be implemented.

3-5. IMPLEMENTATION DETAILS

The syntax analyser is of the recursive descent type. It is implemented as a collection of recursive procedures. Each procedure corresponds to a nonterminal symbol of the LL(1) grammar defining the syntax of the grammar.

The syntax analyser given in Appendix IV is synthesized by writing a procedure for each nonterminal of the grammar given in Appendix III.

The syntax analyser has available a procedure CHECKSYM which has two parameters. The first parameter is the value of the expected symbol and the second parameter is an error number which indicates the error in case the input symbol does not match the expected input symbol. A boolean function TESTSYM is also available to the syntax analyser which returns true in case the current input symbol available in the variable SYM matches the symbol passed to the function as a parameter.

For reporting the error we have a procedure ERROR which has a parameter which is the error number.

```
procedure CHECKSYM (CSYM : SYMBOL; ERR;integer);
  begin if TESTSYM(CSYM) then GETSYM
          else ERROR (ERR)
  end;
```

Similarly, a boolean function TESTSYMINSET is available which returns true if the current input symbol belongs to a set of symbols passed as a parameter to this function. TESTSYMINSET is used when a particular action of the parser depends upon the input symbol being one of the symbols of a set.

The body of the syntax analyser consists of a call to the procedure GETSYM to initialize SYM, followed by a call to procedure PROGRAMHEADER, then a call to BLOCK followed by a check TESTSYM (PERIOD). This corresponds to the production for the starting nonterminal <program>.

<program> ::= <program><programheader> <block>.

Now we will give an example of a parser procedure.

Example 3-1

From Appendix III, Rule (51), we have

```
<STATEMENT> ::= being <STATLIST> end
              | if <IFSTAT>
              | while <WHILESTAT>
              | repeat <REPEATSTAT>
              |<IDENTIFIER><OTHERSTAT>
```

The procedure appears as

```
procedure STATEMENT;
  being
   if TESTSYM (BEGINSYM) then
    begin GETSYM; STATLIST;
      CHECKSYM (ENDSYM, 13)
    end
   else if TESTSYM (IFSYM) then
          begin GETSYM; IFSTAT
          end
        else if TESTSYM (WHILESYM) then
                begin GETSYM; WHILESTAT
                end
             else if TESTSYM (REPEATSYM) then
                     begin GETSYM; REPEATSTAT
                     end
                  else if TESTSYM (IDENT) then
                          begin GETSYM; OTHERSTAT
```

REMARKS

Though the syntax of the language is nontrivial, the task of constructing a syntax analyser, once we have the LL(1) grammar, is very simple.

CHAPTER 4

SYNTAX ERROR RECOVERY

4-1. WHY ERROR RECOVERY

The syntax can be presisely defined using context-free grammars. Therefore, any error in context-free syntax can be detected by the syntax analyser. Error recovery is desirable because compilation should be completed on flawed programs at least through the syntax analysis phase, so that as many errors as possible can be detected in one compilation. Recursive descent parsers have the valid prefix property, i.e., they announce error as soon as a prefix of the input has been seen for which there is no valid continuation.

A good error recovery scheme should have the following properties.

(a) it should pick up immediately after the detection of an error;

(b) should not emit unjustified error messages, and

(c) no error should escape its detection.

The recovery scheme used is the 'panic mode' error recovery (Ammann78, Backhouse 79). In panic mode, the parser discards input symbols on encountering an error till it finds a 'synchronizing' symbol. A synchronizing symbol is a symbol which can legally follow the current state of the parse. Control of the parser is then allowed to proceed to the point at which the symbol is expected and the parsing resumed.

4-2. IMPLEMENTATION DETAIL

The error recovery is based on a procedure having two parameters. One of them is FSYS1 which is the set of symbols which are expected to follow the current state of the parse. The other set of symbols FSYS2

denotes the symbols which can synchronize the action of the parser with the input symbol in case the SYM is incompatible with the current state of the parse.

The two symbol sets are of

```
type SYMSET = set of SYMBOL
```

and the procedure reads

```
procedure TEST(FSYS1, FSYS2 : SYMSET; ERR:integer);
  begin if not TESTSYMINSET (FSYS1) then
        begin ERROR(ERR); FSYS1  := FSYS1+FSYS2;
          while not TESTSYMINSET (FSYS1) do GETSYM
        end
  end;
```

FORMALISM BEHIND THE PARAMETERS FSYS1 AND FSYS2 OF THE PROCEDURE TEST

The parameter FSYS1 contains the FIRST symbols of the nonterminal for which parsing is to be done whereas FSYS2 contains the FOLLOW symbols of the nonterminal.

The definition of the sets FIRST and FOLLOW are:

For a context-free grammar $G = (N,T,P,S)$ with no useless productions, the FIRST and FOLLOW on $N \cup T$ is defined as given in Definition 3-4.

The set FSYS2 contains the synchronizing symbols. The procedure TEST adjusts the input string after detection of an error.

Now we come to the rules for implementing this error recovery scheme:

(1) Every parser procedure has parameter (FSYS of type SYMSET), through which the procedure is informed of the symbols which should not be skipped over during the call. The initial value of FSYS, passed to the procedure which corresponds to the starting non-terminal of the language, is the empty set. Subsequently, when procedure pB is called from within pA then the value of FSYS passed to pB is the union of the value of FSYS passed to pA and the set of terminal symbols which are tested within pA following the call to pB, i.e., the

subset of FOLLOW(B) which is derived from the production of A which corresponds to the selected path in procedure pA.

(2) TEST is called at the beginning and end of each nonterminal procedure except when the logic of the program makes the call unnecessary. If procedure pA is called unconditionally and if pA does not immediately call another procedure pB, then TEST is called on entering pA.

(3) Test is called before leaving procedure pA unless the last action of pA was a call to a procedure pB.

From the above set of rules, we see that the handling of the syntactic errors is ultimately done by the called procedure. But the the calling procedure has full control over the error recovery in the called procedure due to the value of FSYS it passed to the called procedure. The value of FSYS passed to a procedure depends upon the syntax of the nonterminal.

Now we can mechanically with the help of the above rules enrich the syntax analyser developed in Chapter 3 to recover from errors. The syntax analyser with error recovery is shown in Appendix V.

Now we present an example which illustrates the points made above.

<u>Example 4-1</u>

From Appendix III, the syntax from Rule (3-3) is

```
<EXPRESSION>  ::=  <SIMEXP>
                  |<SIMEXP> <RELOPS> <SIMEXP>
```

The corresponding procedure is

```
procedure EXPRESSION (FSYS : SYMSET);
  begin SIMEXP (FSYS + RELOPSYMS);
    if TESTSYMINSET (RELOPSYMS) then
      begin GETSYM;
        SIMEXP (FSYS)
      end
  end;
```

REMARKS

(i) Since SIMEXP is called from EXPRESSION, the set of follow symbols passed to SIMEXP include the terminal symbols which are tested after the call.

(ii) No call to procedure TEST at the beginning and at the end of the procedure exists because the logic of this procedure makes the call unnecessary

4-3. ADVANTAGES OF THE ERROR RECOVERY SCHEME

This scheme of error recovery has the following advantages:

(a) It is simple to implement

(b) It can never get into a loopbecause any recovery action eventually results in an input symbol being consumed or the implicit stack (the suspended procedures) being shortened if the end of the input has been reached.

CHAPTER 5

CONTEXT SENSITIVE ANALYSIS

5-1. INTRODUCTION

The context-free syntax of a language is inadequate for it cannot specify the context-sensitive features. For example, the context-sensitive features of PASCAL like operator/operand type compatibility, type equivalence, identifier scope and the declaration-before-use rule have to be incorporated in the language definition in English (Jensen and Wirth 74). However, there do exist formalisms that are addressed to such tasks. The best known formalism is that of context-sensitive grammars. Whereas the Context Sensitive Grammars are adequate for the formal specification of the task, they are not well-suited for our purpose on two accounts:

(i) A parse of a sentence in a context sensitive language cannot be simply depicted by a parse-tree. It will have to be represented by a complex graph that is messy to draw and comprehend. This lack of simplicity in conceptualisation has been a major cause for the disuse of context sensitive Grammars in the formal specification of programming languages.

(ii) There is no simple mechanism of extension of CFGs, on which our entire developmental effort has been predicated, which absorbs the context sensitive aspects and leads to a context sensitive grammar.

For these reasons, there has been a strong tendency to preserve the context-free core of the formal specification in the extensions proposed.

A well known extension of CFGs to handle other than context-free aspects is that of Attribute-Grammars (AG) (Knuth 68). This extension is powerful enough to define not only the context sensitive aspects of a programming language but also its semantics.

Our efforts towards a formal specification of the Context-Sensitive aspect of a programming language start with the use of AGs. As AGs are more general than necessary for our purpose, we look for a two fold restriction:

(i) We would like a direct relationships between the leftmost-derivation process and the evaluation of attributes.

(ii) We would like to restrict the attribute domains such that we can consider the definition of context-sensitive aspects of the language in question but not necessarily its semantics.

The first restriction is made possible through the use of L-Attributed Grammars (Lewis Rosenkrantz and Stearns 77) and the second through the use of Extended Attributes Grammars (Watt and Madsen 77,79). We look for a synthesis of these two systems to satisfy our purpose.

5-2. ATTRIBUTE GRAMMARS

An Attribute Grammar may be defined as

$$AG = \langle N, T, SA, IA, PA, S \rangle$$

where N,T and S are as in CFGs and
SA is a set of synthesized Attribute Names
IA is a set of Inherited Attribute Names
PA is a set of Attributed Productions of the form
$\langle\langle A, IA^*, SA^* \rangle, \langle V^*, AER^* \rangle\rangle$
where $A \in N$
and AER are Attribute Evaluation Rules expressed in some algorithmic language.

The interpretation of a AG definition is the following:

Construct the parse tree of a sentence by using the CFG embedded in the AG. Annotate each non-terminal in the tree by the corresponding lists of Inherited and Synthesized Attribute Names. This information is available from the first element of the pairs in PA. Associate the corresponding AERs also with the nonterminal node. Now find an order of evaluation of

the AERs such that all the inherited and synthesized Attribute Names that annotate the internal nodes of the true have defined values.

A classical difficulty concerning AGs, pointed out by Knuth in his definitive paper (Knuth 68) is that the AERs may be circular, a fact that can be algorithmically detected.

5-3. L-ATTRIBUTED GRAMMARS (LEWIS RESENKRANTZ and STEARNS 77)

Several restrictions may be imposed on AERs such that

(i) non-circularity is guaranteed
(ii) an order of evaluation of AERs can be known in advance.

As we suppose that AGs are to be used in conjunction with some parsing technique, the order of evaluation of the AERs can be tied to the order of traversal of the parse tree that is effected by the parser. Restricting the AERs such that no undefined attributes (inherited or synthesized) exist at this point of their evaluation (dictated by the order of traversa will rule out circularity.

L-Attributed Grammars result from restrictions of order of traversal that arise from Recursive Descent Parsing.

5-4. EXTENDED ATTRIBUTE GRAMMARS (EAG)

The difference between AGs and EAGs is that the attribute positions in an EAG rule may be occupied by attribute expressions rather than by jus attribute variables (Watt and Madsen 79).

An EAG is defined as (Watt and Madsen 79)

$$G = \langle D,V,Z,B,R \rangle$$

where D = (D1,D2, ..., f1,f2, ...) is an algebraic structure with domains D1,D2, ..., and (partial) functions f1,f2,... operating on Cartesian products of these domains. Each object in one of these domains is called an attribute.

V is the vocabulary of G, a finite set of symbols which is partitioned into the nonterminal vocabulary V_N and the terminal vocabulary V_T.

Z is the distinguished nonterminal of G, i.e., the axiom symbol.

It is assumed that Z has no attribute-position and that no terminal symbol has any inherited attribute positions.

B is a finite collection of attribute variables. Each variable has a fixed domain from D.

R is a finite set of production rule forms.

The interpretation of a EAG definition is the following:

Let $F ::= F_1, \dots, F_m$ be a rule. Take a variable x which occurs in this rule, select any attribute a in the domain of x, and systematically substitute a for x throughout the rule. Repeat such substitutions until no variables remain, then evaluate all the attribute expressions. Provided all the attribute expressions have defined values, this yields a production rule:

$$A ::= A_1 \dots A_m$$

where $m >= 0$ and $A, A_1, A_2, \dots, A_m$ are attribute symbols, A being an attributed nonterminal.

A terminal production of A is a production of A which consists entirely of attributed terminals.

A sentence of G is a terminal production of Z.

The language generated by G is the the set of all sentences of G.

5-5. ATTRIBUTE DOMAIN TYPES AND THE OPERATIONS DEFINED ON THEM

The domain types, used in the EAG definition for PASCAL (Watt and Madsen 79), defined are the following:

Cartisian Products

If T_1, ..., T_n are domains and g_1, ..., g_n are distinct names, then

$$p = (g_1 : T_1; \ldots; g_n : T_n)$$

is a Cartesian product with field selectors g_1, ..., g_n.

The composition function for the Cartisian product P is: for every a_1 in T_1, ..., and every a_n in T_n, $(a_1, \ldots, a_n)$ is in P.

Discriminated Unions

If T_1, ..., T_n are domains (or Cartisian products of domains) and g_1, ..., g_n are distinct names then

$$U = (g_1(T_1) | \ \ldots \ | g_n(T_n))$$

is a discriminated union with selectors g_1, ..., g_n.

For every $i = 1, \ldots, n$, and for every a_i in T_i, $g_i(a_i)$ is in U. These g_i are the composition functions for the discriminated union U.

Sets

If D is a domain, then

$$S = \text{powerset } D,$$

is the domain of subsets of D.

The operations defined are union ($\cup$) and test for membership ($\in$) and disjoint union ($\uplus$). For each s_1 and s_2 in S, $s_1 \uplus s_2$ is the union of s_1 and s_2 if s_1 and s_2 are disjoint, but is undefined otherwise.

Maps

If D and R are domains, then

$$M = D \rightarrow R$$

is the domain of (partial) maps from D to R.

For every d in D and m in M, m[d] either is defined or is undefined.

For each m and m_2 in M, $m_1 \uplus m_2$ is the disjoint union of m_1 and m_2:

$$(m_1 \uplus m_2)\ [d] = \begin{array}{l} \text{if } m_1\ [d] \text{ is defined} \\ \text{then } m_1\ [d] \\ \text{else } m_2\ [d] \end{array}$$

For each m_1 and m_2 in M, $m_1 \setminus m_2$ is the map m_1 overridden by m_2; i.e.,

$$(m_1 \setminus m_2)\ [d] \ = \begin{array}{l} \text{if } m_2\ [d] \text{is undefined} \\ \text{then } m_1 [d] \\ \text{else } m_2 [d] . \end{array}$$

Sequences

If D is a domain, then

$$S = D^*$$

is the domain of sequences of elements of D.

If s_1 and s_2 are in S, then $s_1 \frown s_2$ denotes the sequence obtained by concatenating s_1 and s_2.

The domains defined are, for example

Environ = Name $\rightarrow$ Mode

Mode = (kind : Kind, type : Type)

Kind = (const
|type
|var
⋮
|field)

Consider the EAG definition for <constant definition list>

<Constant definition list ↓ NONLOCALS ↓ LOCALS1 ↑ LOCALS2>

::= <Constant definition ↓NONLOCALS ↓LOCALS1 ↑ LOCALS2> ";"

| <constant definition list ↓ NONLOCALS ↓ LOCALS1 ↑ LOCALS> ";"

<constant definition ↓ NONLOCALS ↓ LOCALS ↑ LOCALS2>

The attribute variables used in the above production all belong to the domain Environ. The synthesized attribute positions are denoted by ↑ whereas the inherited attribute positions are denoted by ↓ .

The EAG definition of PASCAL given is suited for LR parsing since there is left-recursion involved.

5-6. L-EXTENDED ATTRIBUTE GRAMMAR (L-EAG)

Our parsing strategy of top-down left-to-right parse tree traversal makes the EAG definition available not suitable for implementation of the context-sensitive analysis phase.

The following two conditions can help mould the existing EAG formalism to be more useful for our purpose.

(i) We propose that the productions of the EAG be restricted such that no left-recursion is allowed.

(ii) Then we impose the restriction of L-attribute grammars of attribute evaluation to get the L-EAG.

For example, the earlier EAG production can be written in L-EAG as

<constant definition list ↓NONLOCALS ↓LOCALS1 ↑LOCALS2> ::=

<Constant definition ↓NONLOCALS ↓LOCALS1 ↑LOCALS2> ";"

| <constant definition ↓NONLOCALS LOCALS1↑ LOCALS > ";"

<constant definition list ↓NONLOCALS ↓LOCALS ↑LOCALS2>

Now we stipulate the rules which transform an L-EAG for PASCAL FOR implementation.

5-7. IMPLEMENTATION RULES FOR CONTEXT-SENSITIVE ANALYSIS USING L-EAG DEFINITIONS

The recursive descent syntax analyser (Appendix V) can be enriched for context-sensitive analysis by the following rules:

(1) Implementing the domains of the attribute variables and the operations defined upon them. This is a data structuring problem.

(2) For each attribute-position of a nonterminal, introduce parameters to the corresponding procedure. Since inherited attributes convey information down the parse tree, the parameters corresponding to inherited attribute positions can be value parameters. Whereas, <u>var</u> parameters are included for synthesized attribute positions since the information is passed up the parse tree.

(3) Evaluate the attribute expressions within procedure at the end of each path representing a production.

(4) Introduce local variables for preserving the inherited attributes within a procedure and also to construct synthesized attributes local to the procedure.

The L-EAG definition of PASCAL was attempted by us. However, due to some parts of it concerning type declaration and procedure declarations being incomplete, it has not been included.

CHAPTER 6

CONCLUSIONS

An attempt has been made to develop systematically a language processor with the techniques available in the literature with the goal of formalism at every stage of the development.

This experiment of formalisation-before-development has been found to be usable and effective.

Further work in this direction may include the complete L-EAG definition of PASCAL. Also the problem of transformation, the last phase of the development, poses a tough task with the questions of semantic equivalence and compiler correctness.

REFERENCES

1. Aho and Ullmann 76: Aho, A.H., Ullman, J.D., "Principles of Compiler Design", Addison, Wesley, 1977.

2. Ammann, 78 : Ammann, U., "Error Recovery in Recursive Descent Parsers", IRIA Course Proceedings, 1978

3. Backhouse, 79: Backhouse, R.C., "Syntax of Programming Languages Theory and Practice," Prentice-Hall, 1979.

4. Jensen and Wirth, 74 : Jensen, K., Wirth, N., "PASCAL User Manual and Report", Springer 1974.

5. Knuth, 68 : Knuth, D.E., "Semantics of Context-free Languages", Mathematical Theory 2, 1968.

6. Lewis, Rosenkrantz and Streams 77 : Lewis, Rosenkrantz, Stearns : "Compiler Design Theory", Addison-Wesley, 1978.

7. Manna 74 : Manna, Z., "Mathematical Theory of Computation", McGraw-Hill, 1974.

8. Watt and Madsen, 77 : Watt, D.A., and Madsen, O.L., "Extended Attribute Grammars", July 1977. Univ. of Glasgow Report No. 10.

9. Watt and Madsen, 79 : Watt, D.A., and Madsen, O.L., "An Extend Attribute Grammar for PASCAL", SIGPLAN Notices, Vol. 14, No. 2, February 1977.

10. Wirth 73 : Wirth, N., "Systematic Programming : An Introduction", Prentice-Hall, 1973.

APPENDIX I

Specification for the LEXICAL ANALYSIS PHASE

```
<LETTER> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X

             a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x
<DIGIT> ::= 0|1|2|3|4|5|6|7|8|9
<LETTER OR DIGIT> ::= <LETTER> | <DIGIT>
<IDENTIFIER> ::= <LETTER> <LETTER OR DIGIT>*
<NUMBER> ::= <DIGIT>*
<INTNUM> ::= <DIGIT> <NUMBER>
<SIGN> ::= + | -
<FRACTION TAIL> ::= <EMPTY>
                  | E <SIGN> <INTNUM>
<FRACTION> ::= <INTNUM> <FRACTION TAIL>
<REALNUM TAIL> ::= . <FRACTION>
                 | E <SIGN> <INTNUM>
<REALNUM> ::= <INTNUM> <REALNUM TAIL>

<RESERVED WORD> ::= and | array | begin | const | div | do |
                    else | end | file | forward | function |
                    if | in | mod | not | of | or | packed |
                    procedure | program | record | repeat |
                    set | then | type | until | var | while
<SINGLE CHARACTER SYMBOL> ::= + | - | * | / | ^ | = | ( |
                              ) | [ | ] | , | ; | : | #
<DOUBLE CHARACTER SYMBOL> ::= >= | <= | <> | :=
<LEXICAL SYMBOL> ::= <IDENTIFIER> | <RESERVED WORD> | <INTNU
                   | <REALNUM> | <SINGLE CHARACTER SYMBOL>
                 | <DOUBLE CHARACTER SYMBOL>
<EMPTY> ::=
```

APPENDIX II

LEXICAL ANALYSIS PROGRAM

```
t
RW = 28;          (* NO. OF KEYOWRDS *)
 = 10;            (* NO. OF SIGNIFICANT CHARS IN IDENTIFIERS *)
AX = 132;         (* MAX LINE LENGTH *)

ERRMESS= ' ****  CONGRATS! YOU WIN !!NO ERRORS DETECTED';
ES='  ERRORS DETECTED ';


MBOL =
 (NUL,IDENT,INTNUM,REALNUM,PLUS,MINUS,TIMES,SLASH,POINTER,
    LPAREN,RPAREN,LBRACKET,RBRACKET,EQSYM,NESYM,LTSYM,EOFSYM,
    LESYM,GTSYM,GESYM,ASSIGN,COMMA,PERIOD,SEMICOLON,COLON,
    STRING,ANDSYM,ORSYM,NOTSYM,DIVSYM,MODSYM,BEGINSYM,ENDSYM,IFSYM,
    THENSYM,ELSESYM,WHILESYM,DOSYM,REPEATSYM,UNTILSYM,CONSTSYM,
    TYPESYM,VARSYM,ARRAYSYM,OFSYM,RECORDSYM,PACKEDSYM,filesym,
    FUNCSYM,PROCSYM,PROGSYM,INSYM,FORWARDSYM,SETSYM);
PHA = packed array [1..AL] of char;
MSET=set of SYMBOL;
ORDARRAY= array [1..NORW] of ALPHA;
MBOLARRAy= array [1..NORW] of SYMBOL;
ARARRAY= array [char] of SYMBOL;
INTLINE= packed array [1..LMAX] of char;
NEPOINTEr= 0..LMAX;
UNTOFERR=1..100;


: ALPHA;
 : char;              (* LAST CHAR READ *)
M : SYMBOL;           (* LAST SYMBOL READ *)
RD : KWORDARRAY;
YM : SYMBOLARRAy;
YM : CHARARRAY;
NE : PRINTLINE;
,LL: LINEPOINTEr;
RCOUNT : COUNTOFERR;

edure HALT;
egin
nd;

edure ERROR(N:integer);
onst
 ERRMES='ERROR  ';
egin
 WRITELN(OUTPUT,ERRMES,N);
 WRITELN(TTY,ERRMES,N);
 ERRCOUNT:=ERRCOUNT+1
nd;

edure NEXTCH;

unction CAPITAL(CH:char):char;
   begin CAPITAL:=CH;
     if ORD(CH)>140B then
        CAPITAL:=CHR(ORD(CH)-40B)
   end;
```

APPENDIX II

LEXICAL ANALYSIS PROGRAM

```
const

  NORW = 28;        (* NO. OF KEYOWRDS *)
  AL = 10;          (* NO. OF SIGNIFICANT CHARS IN IDENTIFIERS *)
  LMAX = 132;       (* MAX LINE LENGTH *)

  NOERRMESS= ' ****  CONGRATS! YOU WIN !!NO ERRORS DETECTED';
  EMES='  ERRORS DETECTED ';

type

  SYMBOL =
    (NUL,IDENT,INTNUM,REALNUM,PLUS,MINUS,TIMES,SLASH,POINTER,
     LPAREN,RPAREN,LBRACKET,RBRACKET,EQSYM,NESYM,LTSYM,EOFSYM,
     LESYM,GTSYM,GESYM,ASSIGN,COMMA,PERIOD,SEMICOLON,COLON,
     STRING,ANDSYM,ORSYM,NOTSYM,DIVSYM,MODSYM,BEGINSYM,ENDSYM,IFSYM,
     THENSYM,ELSESYM,WHILESYM,DOSYM,REPEATSYM,UNTILSYM,CONSTSYM,
     TYPESYM,VARSYM,ARRAYSYM,OFSYM,RECORDSYM,PACKEDSYM,filesym,
     FUNCSYM,PROCSYM,PROGSYM,INSYM,FORWARDSYM,SETSYM);
  ALPHA = packed array [1..AL] of char;
  SYMSET=set of SYMBOL;
  KWORDARRAY= array [1..NORW] of ALPHA;
  SYMBOLARRAy= array [1..NORW] of SYMBOL;
  CHARARRAY= array [char] of SYMBOL;
  PRINTLINE= packed array [1..LMAX] of char;
  LINEPOINTEr= 0..LMAX;
  COUNTOFERR=1..100;

var
  A : ALPHA;
  CH : char;            (* LAST CHAR READ *)
  SYM : SYMBOL;         (* LAST SYMBOL READ *)
  WORD : KWORDARRAY;
  WSYM : SYMBOLARRAy;
  SSYM : CHARARRAY;
  LINE : PRINTLINE;
  CC,LL: LINEPOINTEr;
  ERRCOUNT : COUNTOFERR;


procedure HALT;
   begin
   end;

procedure ERROR(N:integer);
   const
      ERRMES='ERROR  ';
   begin
      WRITELN(OUTPUT,ERRMES,N);
      WRITELN(TTY,ERRMES,N);
      ERRCOUNT:=ERRCOUNT+1
   end;

procedure NEXTCH;

   function CAPITAL(CH:char):char;
      begin CAPITAL:=CH;
        if ORD(CH)>140B then
          CAPITAL:=CHR(ORD(CH)-40B)
      end;
```

```
    begin (*NEXTCH*)
      if CC=LL then
         if EOF(INPUT) then  HALT
         else
          begin LL:=0; CC:=0;
            OUTPUT^:=' ';
            PUT(OUTPUT);
            while not(EOLN(INPUT)) do
             begin LL:=LL+1;
               LINE[LL]:=INPUT^;
               OUTPUT^:=INPUT^;
               PUT(OUTPUT);
               GET(INPUT)
             end;
            PUTLN(OUTPUT);
            LL:=LL+1;
            LINE[LL]:=' ';
            GET(INPUT)
          end;
      CC:=CC+1;
      CH:=CAPITAL(LINE[CC])
    end;

procedure GETSYM;
    var
      I,J,K : integer;

    function LETTER:boolean;
       begin
         if (ORD(CH)>=ORD('A')) and (ORD(CH)<=ORD('Z')) then
            LETTER:=true
         else LETTER:=false
       end;

    function DIGIT:boolean;
       begin
         if (ORD(CH)>=ORD('0')) and (ORD(CH)<=ORD('9')) then
            DIGIT:=true
         else DIGIT:=false
       end;

    procedure PACKWORD;
       begin K:=0;
         while DIGIT or LETTER do
          begin
            if K<AL then
             begin K:=K+1;  A[K]:=CH
             end;
            NEXTCH
          end
       end;

    procedure KEYWORDORId;
       begin
         while K<AL do
          begin K:=K+1;  A[K]:=' '
          end;
         I:=1;  J:=NORW;
         repeat
            K := (I+J) div 2;
            if A<=WORD[K] then J:=K-1;
            if A>=WORD[K] then I:=K+1
         until I>J;
         if I-1>J then SYM:=WSYM[K]
         else SYM:=IDENT
       end;

    procedure NUMBER;
       begin
         while DIGIT do NEXTCH
       end;
```

```
procedure REALNUMBER;
   begin NEXTCH;
     if CH='.' then CH:=':'
     else
        if DIGIT then
         begin NUMBER;
           SYM:=REALNUM;
         end
        else
         begin
           SYM:=NUL;
           ERROR(1);
           GETSYM
         end
   end;

procedure EXPONENTNUm;
   begin NEXTCH;
     if (CH='+') or (CH='-') then NEXTCH;
     if DIGIT then
      begin NUMBER;
        SYM:=REALNUM;
      end
     else
      begin
        SYM:=NUL;
        ERROR(2);
        GETSYM
      end
   end;

procedure INTORREALNum;
   begin NUMBER;
     SYM:=INTNUM;
     if CH='B' then NEXTCH
     else
      begin
        if CH='.' then REALNUMBER;
        if (CH='E') then EXPONENTNUm
      end
   end;

begin
  while CH=' ' do NEXTCH;
  if LETTER then
   begin
     PACKWORD;
     KEYWORDORId
   end
  else
     if DIGIT then INTORREALNum
     else
        if CH='''' then
        repeat
           NEXTCH;
           while CH<>'''' do NEXTCH;
           NEXTCH;
           SYM:=STRING
        until CH<>''''
        else
           if CH='<' then
            begin NEXTCH;
              if CH='>' then
               begin SYM:=NESYM;  NEXTCH
               end
              else
                 if CH='=' then
                  begin SYM:=LESYM;  NEXTCH
                  end
                 else SYM:=LTSYM
            end
```

```
               else
                  if CH='>' then
                   begin NEXTCH;
                     if CH='=' then
                      begin SYM:=GESYM;  NEXTCH
                      end
                     else SYM:=GTSYM
                   end
                  else
                     if CH=':' then
                      begin NEXTCH;
                        if CH='=' then
                         begin SYM:=ASSIGN;  NEXTCH
                         end
                        else SYM:=COLON
                      end
                     else
                        if CH='.' then
                         begin NEXTCH;
                           if CH='.' then
                            begin SYM:=COLON;  NEXTCH
                            end
                           else SYM:=PERIOD
                         end
                        else
                           if CH='(' then
                            begin NEXTCH;
                              if CH='*' then
                               begin NEXTCH;
                                 repeat
                                    while CH<>'*' do NEXTCH;
                                    NEXTCH
                                 until CH=')';
                                 SYM:=NUL;
                                 NEXTCH;
                                 GETSYM
                               end
                              else
                                 SYM:=LPAREN
                            end
                           else
                              if (CH in ['+','-','*','/','^','=','(',
                                 ')','[',']',',',';','#']) then
                               begin
                                 SYM:=SSYM[CH];  NEXTCH
                               end
                              else
                               begin
                                 SYM:=NUL;
                                 ERROR(3);
                                 NEXTCH;
                                 GETSYM
                               end;
   end;
begin
   (* INITIALIZATIONS *)
   WORD[  1]:='AND       ';                      WSYM[  1]:=ANDSYM;
   WORD[  2]:='ARRAY     ';                      WSYM[  2]:=ARRAYSYM;
   WORD[  3]:='BEGIN     ';                      WSYM[  3]:=BEGINSYM;
   WORD[  4]:='CONST     ';                      WSYM[  4]:=CONSTSYM;
   WORD[  5]:='DIV       ';                      WSYM[  5]:=DIVSYM;
   WORD[  6]:='DO        ';                      WSYM[  6]:=DOSYM;
   WORD[  7]:='ELSE      ';                      WSYM[  7]:=ELSESYM;
   WORD[  8]:='END       ';                      WSYM[  8]:=ENDSYM;
   WORD[  9]:='FILE      ';                      WSYM[  9]:=FILESYM;
   WORD[ 10]:='FORWARD   ';                      WSYM[10]:=FORWARDSYM;
   WORD[ 11]:='FUNCTION  ';                      WSYM[11]:=FUNCSYM;
   WORD[ 12]:='IF        ';                      WSYM[12]:=IFSYM;
   WORD[ 13]:='IN        ';                      WSYM[13]:=INSYM;
   WORD[ 14]:='MOD       ';                      WSYM[14]:=MODSYM;
```

```
  WORD[15]:='NOT       ';                    WSYM[15]:=NOTSYM;
  WORD[16]:='OF        ';                    WSYM[16]:=OFSYM;
  WORD[17]:='OR        ';                    WSYM[17]:=ORSYM;
  WORD[18]:='PACKED    ';                    WSYM[18]:=PACKEDSYM;
  WORD[19]:='PROCEDURE ';                    WSYM[19]:=PROCSYM;
  WORD[20]:='PROGRAM   ';                    WSYM[20]:=PROGSYM;
  WORD[21]:='RECORD    ';                    WSYM[21]:=RECORDSYM;
  WORD[22]:='REPEAT    ';                    WSYM[22]:=REPEATSYM;
  WORD[23]:='SET       ';                    WSYM[23]:=SETSYM;
  WORD[24]:='THEN      ';                    WSYM[24]:=THENSYM;
  WORD[25]:='TYPE      ';                    WSYM[25]:=TYPESYM;
  WORD[26]:='UNTIL     ';                    WSYM[26]:=UNTILSYM;
  WORD[27]:='VAR       ';                    WSYM[27]:=VARSYM;
  WORD[28]:='WHILE     ';                    WSYM[28]:=WHILESYM;

  SSYM['+']:=PLUS;               SSYM[')']:=RPAREN;
  SSYM['-']:=MINUS;              SSYM['[']:=LBRACKET;
  SSYM['*']:=TIMES;              SSYM[']']:=RBRACKET;
  SSYM['/']:=SLASH;              SSYM[',']:=COMMA;
  SSYM['^']:=POINTER;            SSYM[';']:=SEMICOLON;
  SSYM['=']:=EQSYM;              SSYM['#']:=NESYM;
  SSYM['(']:=LPAREN;

  CH:=' ';  CC:=0;  LL:=0;

  GETSYM;
  while not eof(INPUT) do GETSYM
end.
```

```
(********************************)
(*         APPENDIX III         *)
(********************************)

(1)
<PROGRAM> ::= <PROGRAMHEAD> <BLOCK> .

(2)
<PROGRAMHEAD> ::= program <IDENTIFIER> ( <FILELIST> ) ;

(3)
<FILELIST> ::= <IDENTIFIER> {,<IDENTIFIER>}

(4)
<BLOCK> ::= <CONSTDEFPART> <TYPEDEFPART> <VARDEFPART> <FUNORPRO
                begin <STATLIST> end

                (*   CONSTANT DECLARATION AND DEFINITION   *)
                --------------------------------------------

(5)
<CONSTDEFPART> ::= <EMPTY>
           | const <CONSTDEFLIST>

(6)
<CONSTDEFLIST> ::= <CONSTDEF> ;
                 | <CONSTDEF> ; <CONSTDEFLIST>

(7)
<CONSTDEF> ::= <IDENTIFIER> = <CONSTANT>

(8)
<CONSTANT> ::= <STRING>
             | <SIGNEDCONST>

(9)
<SIGNEDCONST> ::= <SIGN> <IDENTIFIER>
                | <SIGN> <INTEGER OR REAL>
                | <IDENTIFIER>
                | <INTEGER OR REAL>

<SIGN> ::= + | -

<INTEGER OR REAL> ::= <INTNUM>
                    | <REALNUM>

                (*   TYPE DECLARATION AND DEFINITION   *)
                ----------------------------------------

(10)
<TYPEDEFPART> ::= <EMPTY>
                | <TYPEDEFLIST>
```

```
(11)
<TYPEDEFLIST> ::= <TYPEDEFINITION> ;
                | <TYPEDEFINITION> ; <TYPEDEFLIST>

(12)
<TYPEDEFINITION> ::= <IDENTIFIER> = <TYPEDEF>

(13)
<TYPEDEF> ::= set <SETYPE>
            | packed <RECARR>
            | ^ <IDENTIFIER>
            | <RECORD OR ARRAY> <RECARR>
            | <SIMPLETYPE>

<RECORD OR ARRAY> ::= record | array

(14)
<SETYPE> ::= of <SIMPLETYPE>

(15)
<RECARR> ::= record <FIELDLIST> end
           | array <ARRAYTYPE>

(16)
<FIELDLIST> ::= <FIELDIDLIST> : <IDENTIFIER>
              | <FIELDIDLIST< : <IDENTIFIER> ; <FIELDLIST>

(17)
<FIELDIDLIST> ::= <IDENTIFIER>
                | <IDENTIFIER> , <FIELDIDLIST>

(18)
<ARRAYTYPE> ::= [ <SIMPLETYPE> { , <SIMPLETYPE> } ] of <TYPEDEF

(19)
<SIMPLETYPE> ::= ( <IDENTLIST> )
               | <STRRING> .. <STRING>
               | <SIGNEDCONST> .. <SIGNEDCONST>

(20)
<IDENTLIST> ::= <IDENTIFIER>
              | <IDENTIFIER> , <IDENTLIST>

                  (*   VARIABLE DECLARATION   *)
                  ------------------------------

(21)
<VARDEFPART> ::= <EMPTY>
               | var <VARDECLIST>

(22)
<VARDECLIST> ::= <VARDECLARATION> ;
               | <VARDECLARATION> ; <VARDECLIST>

(23)
<VARDECLARATION> ::= <IDENTIFIER> : <TYPE IDENTIFIER>
                   | <IDENTIFIER> , <VARDECLARATION>

                  (*   FUNCTION AND PROCEDURE DECLARATIONS   *)
                  ---------------------------------------------

(24)
<FUNORPROCDECL> ::= <EMPTY>
                  | procedure <PROCHEADER> ; forward ; <FUNORPI
                  | procedure <PROCHEADER> ; <BLOCK> ; <FUNORPI
                  | function <FUNCHEADER> ; forward ; <FUNORPR
                  | function <FUNCHEADER> ; <BLOCK> ; <FUNORPR
```

```
(25)
<FUNCHEADER> ::= <IDENTIFIER> <FUNCPARLST> : <TYPE IDENTIFIER>

(26)
<FUNCPARLST> ::= <EMPTY>
               | ( <FUNCPARAMETERS> )

(27)
<FUNCPARAMETERS> ::= <FPARAIDLST> : <TYPE IDENTIFIER>
                                   { ; <FUNCPARAMETERS> }
                   | procedure <IDENTIFIER> <PROCPARLST>
                                   { ; <FUNCPARAMETERS> }
                   | function <IDENTIFIER> <FUNCPARLST> : <IDEN
                                   { ; <FUNCPARAMETERS> }

(28)
<FPARAIDLST> ::= <IDENTIFIER> { , <IDENTIFIER> }

(29)
<PROCHEADER> ::= <DIENTIFIER> <PROCPARLST>

(30)
<PROCPARLST> ::= <EMPTY>
               | ( <PROCPARAMETERS> ).

(31)
<PROCPARAMETERS> ::= <PPARAIDLST> : <TYPE IDENTIFIER>
                                   { ; <PROCPARAMETERS> }
                   | var <PPARAIDLST> : <TYPE IDENTIFIER>
                                   { ; <PROCPARAMETERS> }
                   | procedure <IDENTIFIER> <PROCPARLST>
                                   { ; <PROCPARAMETERS> }
                   | function <IDENTIFIER> <FUNCPARLST> :
                       <TYPE IDENTIFIER> { ; <PROCPARAMETERS>

(32)
<PPARAIDLST> ::= <IDENTIFIER> { , <IDENTIFIER> }

                  (*   EXPRESSIONS, TERM AND FACTOR   *)
                  ------------------------------------

(33)
<EXPRESSION> ::= <SIMEXP>

               | <SIMEXP> <RELOPS> <SIMEXP>

<RELOPS> ::= = | # | < | > | <= | >= | in

(34)
<SIMEXP> ::= <SIGN> <TERMS>
           | <TERMS>

(35)
<TERMS> ::= <TERM>
          | <TERM> <SETOPS> <TERMS>

<SETOPS> ::= + | - | or

(36)
<TERM> ::= <FACTOR>
         | <FACTOR> <MULOPS> <TERM>

<MULOPS> ::= * | / | div | mod | and
```

```
(37)
<FACTOR> ::= <IDENTIFIER> <FUNORVAR>
           | not <FACTOR>
           | ( <EXPRESSION> )
           | [ <EXPLIST> ]
           | [ ]
           | <CONSTANT SYMS>

<CONSTANT SYMS> ::= <INTNUM> | <REALNUM> | <STRING>

(38)
<FUNORVAR> ::= <SELECTOR>
             | <SELECTOR> ( <EXPLIST> )

<SELECTOR> ::= <EMPTY>
             | [ <EXPLIST> ] <SELECTOR>
             | . <FIELD IDENTIFIER> <SELECTOR>
             | <SELECTOR>

(40)
<EXPLIST> ::= <EXPRESSION>
            | <EXPRESSION> , <EXPLIST>

                    (*   STATEMENTS   *)
                    -------------------

(41)
<STATEMENT> ::= begin <STATLIST> end
              | if <IFSTAT>
              | while <WHILESTAT>
              | repeat <REPEATSTAT>
              | <IDENTIFIER> <OTHERSTAT>

(42)
<STATLIST> ::= <STATEMENT>
             | <STATEMENT> ; <STATLIST>

(43)
<IFSTAT> ::= <EXPRESSION> then <STATEMENT>
           | <EXPRESSION> then <STATEMENT> else <STATEMENT>

(44)
<WHILESTAT> ::= <EXPRESSION> do <STATEMENT>

(45)
<REPEATSTAT> ::= <STATLIST> until <EXPRESSION>

(46)
<OTHERSTAT> ::= <SELECTOR> := <EXPRESSION>
              | <SELECTOR> ( <EXPLIST> )
              | <SELECTOR>

(47)
<EMPTY> ::=
```

APPENDIX IV

SYNTAX ANALYSER PROGRAM

```
const

  NORW = 28;         (* NO. OF KEYOWRDS *)
  AL = 10;           (* NO. OF SIGNIFICANT CHARS IN IDENTIFIERS *)
  LMAX = 132;        (* MAX LINE LENGTH *)

  NOERRMESS= ' ****  CONGRATS! YOU WIN !!NO ERRORS DETECTED';
  EMES='  ERRORS DETECTED ';

type

  SYMBOL =
     (NUL,IDENT,INTNUM,REALNUM,PLUS,MINUS,TIMES,SLASH,POINTER,
        LPAREN,RPAREN,LBRACKET,RBRACKET,EQSYM,NESYM,LTSYM,EOFSYM,
        LESYM,GTSYM,GESYM,ASSIGN,COMMA,PERIOD,SEMICOLON,COLON,
        STRING,ANDSYM,ORSYM,NOTSYM,DIVSYM,MODSYM,BEGINSYM,ENDSYM,IFS
        THENSYM,ELSESYM,WHILESYM,DOSYM,REPEATSYM,UNTILSYM,CONSTSYM,
        TYPESYM,VARSYM,ARRAYSYM,OFSYM,FILESYM,RECORDSYM,PACKEDSYM,
        FUNCSYM,PROCSYM,PROGSYM,INSYM,FORWARDSYM,SETSYM);
  ALPHA = packed array [1..AL] of char;
  SYMSET=set of SYMBOL;
  KWORDARRAY= array [1..NORW] of ALPHA;
  SYMBOLARRAy= array [1..NORW] of SYMBOL;
  CHARARRAY= array [char] of SYMBOL;
  PRINTLINE= packed array [1..LMAX] of char;
  LINEPOINTEr= 0..LMAX;
  COUNTOFERR=1..100;

var
  A : ALPHA;
  CH : char;            (* LAST CHAR READ *)
  SYM : SYMBOL;         (* LAST SYMBOL READ *)
  WORD : KWORDARRAY;
  WSYM : SYMBOLARRAY;
  SSYM : CHARARRAY;
  LINE : PRINTLINE;
  CC,LL: LINEPOINTEr;
  ERRCOUNT : COUNTOFERR;

procedure HALT;
   begin
   end;

procedure ERROR(N:integer); extern;

procedure GETSYM; extern;

function TESTSYM(LEX:SYMBOL):boolean;
   begin TESTSYM := LEX=SYM
   end;

function TESTSYMINSet(LEXSET:SYMSET):boolean;
   begin TESTSYMINSet := SYM in LEXSET
   end;
```

```
procedure CHECKSYM(CSYM:SYMBOL;ERR:integer);
   begin
     if TESTSYM(CSYM) then GETSYM
     else ERROR(ERR)
   end;

procedure SIGNEDCONSt;
   begin
     if ((TESTSYM(PLUS)) or (TESTSYM(MINUS))) then GETSYM;
     if TESTSYM(IDENT) then GETSYM
     else
        if ((TESTSYM(INTNUM)) or (TESTSYM(REALNUM))) then GETSYM
        else ERROR(14)
   end;

procedure CONSTANT;
   begin
     if TESTSYM(STRING) then GETSYM
     else SIGNEDCONSt
   end;

procedure CONSTDEF;
   begin
     if TESTSYM(IDENT) then
      begin GETSYM;
        CHECKSYM(EQSYM,4);
        CONSTANT
      end
   end;

procedure CONSTDEFLIst;
   begin CONSTDEF;
     CHECKSYM(SEMICOLON,5);
     if TESTSYM(IDENT) then CONSTDEFLIst
   end;

procedure IDENTLIST;                        (* TYPE DECLARATIONS *)
   begin
     if TESTSYM(IDENT) then GETSYM;
     while TESTSYM(COMMA) do
      begin GETSYM;
        IDENTLIST
      end
   end;

procedure SIMPLETYPE;
   begin
     if TESTSYM(STRING) then
      begin GETSYM;
        CHECKSYM(COLON,6); CHECKSYM(STRING,15)
      end
     else
        if TESTSYM(LPAREN) then
         begin GETSYM;
           IDENTLIST;
           CHECKSYM(RPAREN,8)
         end
        else
         begin SIGNEDCONSt;
           if TESTSYM(COLON) then
            begin GETSYM; SIGNEDCONSt
            end
         end
   end;

procedure TYPEDEF;
   forward;
```

```
procedure ARRAYTYPE;
   begin
     if TESTSYM(LBRACKET) then
      begin GETSYM;
        SIMPLETYPE;
        while TESTSYM(COMMA) do
         begin SIMPLETYPE
         end;
        CHECKSYM(RBRACKET,8);
        CHECKSYM(OFSYM,11);
        TYPEDEF
      end
   end;

procedure FIELDIDLST;
   begin
     if TESTSYM(IDENT) then GETSYM;
     while TESTSYM(COMMA) do
      begin GETSYM;
        IDENTLIST
      end
   end;

procedure FIELDLIST;
   begin FIELDIDLST;
     CHECKSYM(COLON,6);
     CHECKSYM(IDENT,12);
     if TESTSYM(SEMICOLON) then
      begin GETSYM;
        FIELDLIST
      end
   end;

procedure RECARR;
   begin
     if TESTSYM(RECORDSYM) then
      begin GETSYM;
        FIELDLIST;
        CHECKSYM(ENDSYM,13)
      end
     else
      begin GETSYM;
        ARRAYTYPE
      end
   end;

procedure SETYPE;
   begin CHECKSYM(OFSYM,11);
     SIMPLETYPE
   end;

procedure TYPEDEF;
   begin
     if TESTSYM(SETSYM) then
      begin GETSYM;  SETYPE
      end
     else
        if TESTSYM(PACKEDSYM) then
         begin GETSYM;
         end
        else
           if TESTSYM(
            begin GETS
            end
           else
              if (TES
              else SI
   end;
```

```
procedure TYPDEFINITIon;
   begin
     if TESTSYM(IDENT) then
      begin GETSYM;
        CHECKSYM(EQSYM,4);
        TYPEDEF
      end
   end;

procedure TYPEDEFLISt;
   begin TYPDEFINITIon;
     CHECKSYM(SEMICOLON,5);
     if TESTSYM(IDENT) then TYPEDEFLISt
   end;

procedure VARDECL;         (* VARIABLE DECLARATIONS    *)
   begin
     repeat
        IDENTLIST;
        CHECKSYM(COLON,6);
        CHECKSYM(IDENT,12);
        CHECKSYM(SEMICOLON,5)
     until (not TESTSYM(IDENT))  and not TESTSYMINSet(TYPDECL)
   end;

procedure PROCPARLST ;     (* FUNCTION & PROCEDURE DECL*)
  forward;

procedure FUNCPARLST ;
  forward;

procedure PPARAIDLST ;
   begin
     if TESTSYM(IDENT) then
      begin GETSYM;
        while TESTSYM(COMMA) do
         begin GETSYM;
           CHECKSYM(IDENT,12);
         end
      end
   end;

procedure PROCPARAMEters ;
   begin
     if (TESTSYMINSet([IDENT,VARSYM,PROCSYM,FUNCSYM])) then
      begin
        if TESTSYM(VARSYM) then
         begin GETSYM;
           PPARAIDLST;
           CHECKSYM(COLON,6);
           CHECKSYM(IDENT,12)
         end
        else
           if TESTSYM(PROCSYM) then
            begin GETSYM;
              CHECKSYM(IDENT,12);
              PROCPARLST
            end
           else
              if TESTSYM(FUNCSYM) then
               begin GETSYM;
                 CHECKSYM(IDENT,12);
                 FUNCPARLST;
                 CHECKSYM(COLON,6);
                 CHECKSYM(IDENT,12)
               end
              else
```

```
                begin PPARAIDLST;
                  CHECKSYM(COLON,6);
                  CHECKSYM(IDENT,12)
                end
        end;
       while TESTSYM(SEMICOLON) do
        begin GETSYM;
          PROCPARAMEters
        end
     end;


procedure PROCPARLST;
     begin
       if TESTSYM(LPAREN) then
        begin GETSYM;
          PROCPARAMEters;
          CHECKSYM(RPAREN,8)
        end
     end;


procedure PROCHEADER ;
     begin CHECKSYM(IDENT,12);
       PROCPARLST
     end;


procedure FPARAIDLST ;
     begin
       if TESTSYM(IDENT) then
        begin GETSYM;
          while TESTSYM(COMMA) do
           begin GETSYM;
             CHECKSYM(IDENT,12)
           end
        end
     end;


procedure FUNCPARAMEters ;
     begin
       if (TESTSYMINSet([IDENT,PROCSYM,FUNCSYM])) then
        begin
          if TESTSYM(PROCSYM) then
           begin GETSYM;
             CHECKSYM(IDENT,12);
             PROCPARLST
           end
          else
             if TESTSYM(FUNCSYM) then
              begin GETSYM;
                CHECKSYM(IDENT,12);
                FUNCPARLST;
                CHECKSYM(COLON,6);
                CHECKSYM(IDENT,12)
              end
             else
              begin FPARAIDLST;
                CHECKSYM(COLON,6);
                CHECKSYM(IDENT,12)
              end
        end;
       while TESTSYM(SEMICOLON) do
        begin GETSYM;
          FUNCPARAMEters
        end
     end;
```

```
procedure FUNCPARLST;
   begin
     if TESTSYM(LPAREN) then
      begin GETSYM;
        FUNCPARAMEters;
        CHECKSYM(RPAREN,8)
      end
   end;

procedure FUNCHEADER ;
   begin CHECKSYM(IDENT,12);
     FUNCPARLST;
     CHECKSYM(COLON,6);
     CHECKSYM(IDENT,12)
   end;

procedure BLOCK;
  forward;

procedure FUNORPROCDecl;
   begin
     if (TESTSYMINSet( [PROCSYM,FUNCSYM])) then
      begin
        if TESTSYM(PROCSYM) then
         begin GETSYM; PROCHEADER
         end
        else
         begin GETSYM;
           FUNCHEADER
         end;
        CHECKSYM(SEMICOLON,5);
        if TESTSYM(FORWARDSYM) then GETSYM
        else BLOCK;
        CHECKSYM(SEMICOLON,5);
        FUNORPROCDecl
      end
   end;

procedure EXPRESSION;
  forward;

procedure EXPLIST;
   begin
     EXPRESSION;
     if TESTSYM(COMMA) then
      begin GETSYM;
        EXPLIST
      end
   end;

procedure SELECTOR;
   begin
     if TESTSYMINSet(SELECTSYS) then
      begin
        if TESTSYM(LBRACKET) then
         begin GETSYM;
           EXPLIST;
           CHECKSYM(RBRACKET,10)
         end
        else
           if TESTSYM(PERIOD) then
            begin GETSYM;
              CHECKSYM(IDENT,12)
            end
           else
              if TESTSYM(POINTER) then GETSYM;
        SELECTOR
      end
   end;
```

```
procedure FUNORVAR;
   begin SELECTOR;
      if TESTSYM(LPAREN) then
       begin GETSYM;
          EXPLIST;
          CHECKSYM(RPAREN,8)
       end
   end;


procedure FACTOR;
   begin
        if TESTSYM(IDENT) then
         begin GETSYM; FUNORVAR
         end
        else
            if (TESTSYMINSet([INTNUM,REALNUM,STRING])) then GETSYM
            else
                if TESTSYM(NOTSYM) then
                 begin GETSYM; FACTOR
                 end
                else
                    if TESTSYM(LPAREN) then
                     begin GETSYM;
                        EXPRESSION;
                        CHECKSYM( RPAREN,8)
                     end
                    else
                        if TESTSYM(LBRACKET) then
                         begin
                            GETSYM;
                            if not (TESTSYM(RBRACKET)) then
                                EXPLIST;
                            CHECKSYM(RBRACKET,10)
                         end
   end;

procedure TERM;
   begin FACTOR;
      if (TESTSYMINSet([TIMES,SLASH,DIVSYM,MODSYM,ANDSYM])) then
       begin GETSYM; TERM
       end
   end;

procedure TERMS;
   begin TERM;
      if (TESTSYMINSet([PLUS,MINUS,ORSYM])) then
       begin GETSYM;
          TERMS
       end
   end;


procedure SIMEXP;
   begin
      if (TESTSYMINSet([PLUS,MINUS])) then GETSYM;
      TERMS
   end;


procedure EXPRESSION;
   begin SIMEXP;
      if (TESTSYMINSet( RELOPSYMS )) then
       begin GETSYM;
          SIMEXP
       end
   end;
```

```
procedure STATEMENT;
  forward;

procedure STATLIST;
   begin
     STATEMENT;
     if TESTSYM(SEMICOLON) then
      begin GETSYM;
        STATLIST
      end
   end;

procedure IFSTAT;
   begin EXPRESSION;
     CHECKSYM(THENSYM,16);
     STATEMENT;
     if TESTSYM(ELSESYM) then
      begin GETSYM; STATEMENT
      end
   end;

procedure WHILESTAT;
   begin EXPRESSION;
     CHECKSYM(DOSYM,17);
     STATEMENT
   end;

procedure REPEATSTAT;
   begin STATLIST;
     CHECKSYM(UNTILSYM,18);
     EXPRESSION
   end;

procedure OTHERSTAT;
   begin SELECTOR;
     if TESTSYM(ASSIGN) then
      begin GETSYM; EXPRESSION
      end
     else
        if TESTSYM(LPAREN) then
         begin GETSYM;
           EXPLIST;
           CHECKSYM(RPAREN,8)
         end
   end;

procedure STATEMENT;
   begin
        if TESTSYM(BEGINSYM) then
         begin GETSYM; STATLIST;
           CHECKSYM(ENDSYM,13)
         end
        else
           if TESTSYM(IFSYM) then
            begin GETSYM; IFSTAT
            end
           else
              if TESTSYM(WHILESYM) then
               begin GETSYM; WHILESTAT
               end
              else
                 if TESTSYM(REPEATSYM) then
                  begin GETSYM; REPEATSTAT
                  end
                 else
```

```
                    if TESTSYM(IDENT) then
                     begin GETSYM; OTHERSTAT
                     end
   end;

procedure BLOCK;
   begin
     if TESTSYM(CONSTSYM) then
      begin GETSYM; CONSTDEFLISt
      end;
     if TESTSYM(TYPESYM) then
      begin GETSYM; TYPEDEFLISt
      end;
     if TESTSYM(VARSYM) then
      begin GETSYM; VARDECL
      end;
     if TESTSYMINSet( [PROCSYM,FUNCSYM]) then
        FUNORPROCDecl;
     CHECKSYM(BEGINSYM,19);
     STATLIST;
     CHECKSYM(ENDSYM,13)
   end;

procedure FILELIST;
   begin
     if TESTSYM(IDENT) then
      begin GETSYM;
        while TESTSYM(COMMA) do
         begin GETSYM;
           CHECKSYM(IDENT,12)
         end
      end
   end;

procedure PROGRAMHEAd;
   begin
     if TESTSYM(PROGSYM) then
      begin GETSYM;
        if TESTSYM(IDENT) then
         begin GETSYM;
           if TESTSYM(LPAREN) then
            begin GETSYM;
              FILELIST;
              if TESTSYM(RPAREN) then
               begin GETSYM;
                 CHECKSYM(SEMICOLON,5)
               end
              else ERROR(8)
            end
           else ERROR(7)
         end
        else ERROR(12)
      end
     else ERROR(20)
   end;


begin
  (* INITIALIZATION OF TABLES USED FOR LEXICAL ANALYSIS *)

  FACBEGSYM :=[LPAREN,NOTSYM,INTNUM,REALNUM,IDENT,STRING,LBRACKET];
  SIMPTYBEGSym :=[STRING,LPAREN,PLUS,MINUS,IDENT,INTNUM,REALNUM];
  SELECTSYS :=[POINTER,PERIOD,LBRACKET];
  TYPEBEGSYM :=[PLUS,MINUS,INTNUM,REALNUM,STRING,IDENT,LPAREN,POINT
  ,PACKEDSYM,ARRAYSYM,RECORDSYM,SETSYM];
  TYPDECL :=[RECORDSYM,ARRAYSYM,SETSYM];
  RELOPSYMS :=[EQSYM,NESYM,LTSYM,LESYM,GTSYM,GESYM,INSYM];
```

```
  GETSYM;
  PROGRAMHEAd;
  BLOCK;
  if not TESTSYM(PERIOD) then ERROR(21);
  if ERRCOUNT<>0 then
   begin
      WRITELN; WRITELN;
      WRITE (OUTPUT,ERRCOUNT);
      WRITE (OUTPUT,EMES);
      WRTTE (TTY,ERRCOUNT);
      WRTTE (TTY,EMES)
   end
  else
   begin WRITELN (OUTPUT,NOERRMESS);
      WRITELN (TTY,NOERRMESS)
   end
end.
```

```
(*
                 APPENDIX V
                 ----------

        SYNTAX ANALYSER WITH CONTEXT-FREE ERROR RECOVERY
        ------------------------------------------------*)

 const

   NOERRMESS= ' ****  CONGRATS! YOU WIN !!NO ERRORS DETECTED';
   EMES='  ERRORS DETECTED ';

 type

   SYMBOL =
      (NUL,IDENT,INTNUM,REALNUM,PLUS,MINUS,TIMES,SLASH,POINTER,
        LPAREN,RPAREN,LBRACKET,RBRACKET,EQSYM,NESYM,LTSYM,EOFSYM,
        LESYM,GTSYM,GESYM,ASSIGN,COMMA,PERIOD,SEMICOLON,COLON,
        STRING,ANDSYM,ORSYM,NOTSYM,DIVSYM,MODSYM,BEGINSYM,ENDSYM,IF
        THENSYM,ELSESYM,WHILESYM,DOSYM,REPEATSYM,UNTILSYM,CONSTSYM,
        TYPESYM,VARSYM,ARRAYSYM,OFSYM,FILESYM,RECORDSYM,PACKEDSYM,
        FUNCSYM,PROCSYM,PROGSYM,INSYM,FORWARDSYM,SETSYM);
   SYMSET=set of SYMBOL;

 var
   SYM : SYMBOL;         (* LAST SYMBOL READ *)
   ERRCOUNT:integer;

   CONSTBEGSYm,SIMPTYBEGSym,SELECTSYS,TYPEBEGSYM,MULOPSYMS,
   TYPDECL,DECLBEGSYM,STATBEGSYM,FACBEGSYM,RELOPSYMS:  SYMSET;

 procedure ERROR(N:integer); extern;

 function TESTSYM(TSYM:SYMBOL):boolean; extern;
 function TESTSYMINSET(SYMBOLSET:SYMSET):boolean; extern;

 procedure GETSYM; extern;

 procedure TEST (S1,S2:SYMSET;N:integer);
    begin
      if not TESTSYMINSet(S1) then
       begin ERROR(N); S1 := S1 + S2;
          while not TESTSYMINSet(S1) do GETSYM
       end
    end;

 procedure CHECKSYM(CSYM:SYMBOL;ERR:integer);
    begin
      If TESTSYM(CSYM) then GETSYM
      else ERROR(ERR)
    end;

 procedure SIGNEDCONSt(FSYS:SYMSET);
    begin
      if ((TESTSYM(PLUS)) or (TESTSYM(MINUS)))
      then GETSYM;
      if TESTSYM(IDENT) then GETSYM
      else
         if ((TESTSYM(INTNUM)) or (TESTSYM(REALNUM)))
         then GETSYM
         else TEST([],FSYS,14)
    end;

 procedure CONSTANT(FSYS:SYMSET);
    begin
      TEST(CONSTBEGSYm,FSYS,14);
      if TESTSYM(STRING) then GETSYM
      else SIGNEDCONSt(FSYS)
    end;
```

```
procedure CONSTDEF(FSYS:SYMSET);
   begin TEST([IDENT],FSYS,12);
     if TESTSYM(IDENT) then
      begin GETSYM;
        CHECKSYM(EQSYM,4);
        CONSTANT(FSYS+[SEMICOLON,IDENT])
      end
   end;

procedure CONSTDEFLIst(FSYS:SYMSET);
   begin CONSTDEF(FSYS+[SEMICOLON]);
     CHECKSYM(SEMICOLON,5);
     if TESTSYM(IDENT) then CONSTDEFLIst(FSYS);
     TEST(FSYS,[],104)
   end;

procedure IDENTLIST(FSYS:SYMSET);                    (* TYPE DECLARATIONS
   begin
     TEST([IDENT],FSYS,12);
     if TESTSYM(IDENT) then GETSYM;
     while TESTSYM(COMMA) do
      begin GETSYM;
        IDENTLIST(FSYS+[COMMA])
      end
   end;

procedure SIMPLETYPE(FSYS:SYMSET);
   begin
     TEST(SIMPTYBEGSym,FSYS,101);
     if TESTSYMINSet(SIMPTYBEGSym) then
      begin
        if TESTSYM(STRING) then
         begin GETSYM;
           CHECKSYM(COLON,6); CHECKSYM(STRING,15)
         end
        else
           if TESTSYM(LPAREN) then
            begin GETSYM;
              IDENTLIST(FSYS+[RPAREN]);
              CHECKSYM(RPAREN,8)
            end
           else
            begin SIGNEDCONSt(FSYS+[COLON]);
              if TESTSYM(COLON) then
               begin GETSYM; SIGNEDCONSt(FSYS)
               end
            end
      end
   end;

procedure TYPEDEF(FSYS:SYMSET);
  forward;

procedure ARRAYTYPE(FSYS:SYMSET);
   begin TEST([LBRACKET],FSYS,9);
     if TESTSYM(LBRACKET) then
      begin GETSYM;
        SIMPLETYPE(FSYS+[COMMA,RBRACKET]);
        while TESTSYM(COMMA) do
         begin SIMPLETYPE(FSYS+[COMMA,RBRACKET])
         end;
        CHECKSYM(RBRACKET,10);
        CHECKSYM(OFSYM,11);
        TYPEDEF(FSYS)
      end
   end;
```

```
procedure FIELDIDLST(FSYS:SYMSET);
   begin
      TEST([IDENT],FSYS,12);
      if TESTSYM(IDENT) then GETSYM;
      while TESTSYM(COMMA) do
       begin GETSYM;
          IDENTLIST(FSYS+[COMMA])
       end
   end;

procedure FIELDLIST(FSYS:SYMSET);
   begin FIELDIDLST(FSYS+[COLON]);
      CHECKSYM(COLON,6);
      CHECKSYM(IDENT,12);
      if TESTSYM(SEMICOLON) then
       begin GETSYM;
          FIELDLIST(FSYS)
       end
      else TEST(FSYS,[],102)
   end;

procedure RECARR(FSYS:SYMSET);
   begin
      if TESTSYM(RECORDSYM) then
       begin GETSYM;
          FIELDLIST(FSYS+[ENDSYM]);
          CHECKSYM(ENDSYM,13)
       end
      else
       begin GETSYM;
          ARRAYTYPE(FSYS)
       end
   end;

procedure SETYPE(FSYS:SYMSET);
   begin CHECKSYM(OFSYM,11);
      SIMPLETYPE(FSYS)
   end;

procedure TYPEDEF;
   begin
      TEST(TYPEBEGSYM,FSYS,103);
      if TESTSYMINSet(TYPEBEGSYM) then
       begin
          if TESTSYM(SETSYM) then
           begin GETSYM;  SETYPE(FSYS)
           end
          else
             if TESTSYM(PACKEDSYM) then
              begin GETSYM; RECARR(FSYS)
              end
             else
                if TESTSYM(POINTER) then
                 begin GETSYM; CHECKSYM(IDENT,12)
                 end
                else
                   if (TESTSYMINSet([RECORDSYM,ARRAYSYM]))
                    then   RECARR(FSYS)
                   else SIMPLETYPE(FSYS);
         TEST(FSYS,[],104)
       end
   end;
```

```
procedure TYPDEFINITIon(FSYS:SYMSET);
   begin TEST([IDENT],FSYS,12);
     if TESTSYM(IDENT) then
      begin GETSYM;
        CHECKSYM(EQSYM,4);
        TYPEDEF(FSYS+[SEMICOLON,IDENT])
      end
   end;

procedure TYPEDEFLISt(FSYS:SYMSET);
   begin TYPDEFINITIon(FSYS+[SEMICOLON]);
     CHECKSYM(SEMICOLON,5);
     if TESTSYM(IDENT) then TYPEDEFLISt(FSYS);
     TEST(FSYS,[],105)
   end;


procedure VARDECLARATION(FSYS:SYMSET);
   begin TEST([IDENT],FSYS,12);
     if TESTSYM(IDENT) then
      begin GETSYM;
        if TESTSYM(COLON) then
         begin GETSYM;
           CHECKSYM(IDENT,12)
         end
        else
         begin CHECKSYM(COMMA,22);
           VARDECLARATION(FSYS)
         end
      end
   end;

  procedure VARDECLIST(FSYS:SYMSET);
     begin VARDECLARATION(FSYS);
       CHECKSYM(SEMICOLON,5);
       if TESTSYM(IDENT) then
         VARDECLIST(FSYS)
     end;

 procedure PROCPARLST (FSYS:SYMSET);     (* FUNCTION & PROCEDURE DECL*
   forward;

 procedure FUNCPARLST (FSYS:SYMSET);
   forward;

 procedure PPARAIDLST (FSYS:SYMSET);
    begin TEST([IDENT],FSYS,12);
      if TESTSYM(IDENT) then
       begin GETSYM;
         while TESTSYM(COMMA) do
          begin GETSYM;
            CHECKSYM(IDENT,12)
          end
       end;
      TEST(FSYS,[],106)
    end;
```

```
procedure PROCPARAMEters (FSYS:SYMSET);
   begin TEST([IDENT,VARSYM,PROCSYM,FUNCSYM],FSYS,107);
     if (TESTSYMINSet([IDENT,VARSYM,PROCSYM,FUNCSYM])) then
      begin
        if TESTSYM(VARSYM) then
         begin GETSYM;
           PPARAIDLST(FSYS+[COLON,IDENT]);
           CHECKSYM(COLON,6);
           CHECKSYM(IDENT,12)
         end
        else
           if TESTSYM(PROCSYM) then
            begin GETSYM;
              CHECKSYM(IDENT,12);
              PROCPARLST(FSYS)
            end
           else
              if TESTSYM(FUNCSYM) then
               begin GETSYM;
                 CHECKSYM(IDENT,12);
                 FUNCPARLST(FSYS+[COLON,IDENT]);
                 CHECKSYM(COLON,6);
                 CHECKSYM(IDENT,12)
               end
              else
               begin PPARAIDLST(FSYS+[COLON,IDENT]);
                 CHECKSYM(COLON,6);
                 CHECKSYM(IDENT,12)
               end
      end;
     while TESTSYM(SEMICOLON) do
      begin GETSYM;
        PROCPARAMEters(FSYS)
      end
   end;

procedure PROCPARLST;
   begin TEST([LPAREN,SEMICOLON],FSYS,108);
     if TESTSYM(LPAREN) then
      begin GETSYM;
        PROCPARAMEters(FSYS+[RPAREN]);
        CHECKSYM(RPAREN,8)
      end;
     TEST(FSYS,[],108)
   end;

procedure PROCHEADER (FSYS:SYMSET);
   begin CHECKSYM(IDENT,12);
     PROCPARLST(FSYS)
   end;

procedure FPARAIDLST (FSYS:SYMSET);
   begin TEST([IDENT],FSYS,12);
     if TESTSYM(IDENT) then
      begin GETSYM;
        while TESTSYM(COMMA) do
         begin GETSYM;
           CHECKSYM(IDENT,12)
         end
      end;
     TEST(FSYS,[],109)
   end;
```

```
procedure FUNCPARAMEters (FSYS:SYMSET);
   begin TEST([IDENT,PROCSYM,FUNCSYM],FSYS,110);
     if (TESTSYMINSet([IDENT,PROCSYM,FUNCSYM])) then
      begin
        if TESTSYM(PROCSYM) then
         begin GETSYM;
           CHECKSYM(IDENT,12);
           PROCPARLST(FSYS)
         end
        else
           if TESTSYM(FUNCSYM) then
            begin GETSYM;
              CHECKSYM(IDENT,12);
              FUNCPARLST(FSYS+[COLON,IDENT]);
              CHECKSYM(COLON,6);
              CHECKSYM(IDENT,12)
            end
           else
            begin FPARAIDLST(FSYS+[COLON,IDENT]);
              CHECKSYM(COLON,6);
              CHECKSYM(IDENT,12)
            end
      end;
     while TESTSYM(SEMICOLON) do
      begin GETSYM;
        FUNCPARAMEters(FSYS)
      end
   end;

procedure FUNCPARLST;
   begin TEST([LPAREN,COLON],FSYS,111);
     if TESTSYM(LPAREN) then
      begin GETSYM;
        FUNCPARAMEters(FSYS+[RPAREN]);
        CHECKSYM(RPAREN,8)
      end;
     TEST(FSYS,[],112)
   end;

procedure FUNCHEADER (FSYS:SYMSET);
   begin CHECKSYM(IDENT,12);
     FUNCPARLST(FSYS+[COLON,IDENT]);
     CHECKSYM(COLON,6);
     CHECKSYM(IDENT,12);
     TEST(FSYS,[],113)
   end;

procedure BLOCK(FSYS:SYMSET);
  forward;

procedure FUNORPROCDecl(FSYS:SYMSET);
   begin
     TEST(FSYS,[],114);
     if (TESTSYMINSet( [PROCSYM,FUNCSYM])) then
      begin
        if TESTSYM(PROCSYM) then
         begin GETSYM; PROCHEADER(FSYS+[SEMICOLON])
         end
        else
         begin GETSYM;
           FUNCHEADER(FSYS+[SEMICOLON])
         end;
        CHECKSYM(SEMICOLON,5);
        if TESTSYM(FORWARDSYM) then GETSYM
        else BLOCK(FSYS+[SEMICOLON]);
        CHECKSYM(SEMICOLON,5);
        FUNORPROCDecl(FSYS+[FUNCSYM,PROCSYM])
      end
   end;
```

```
procedure EXPRESSION(FSYS:SYMSET);
  forward;

procedure EXPLIST(FSYS:SYMSET);
    begin
      EXPRESSION(FSYS+[COMMA]);
      if TESTSYM(COMMA) then
       begin GETSYM;
         EXPLIST(FSYS)
       end;
      TEST(FSYS,[],115)
    end;

procedure SELECTOR(FSYS:SYMSET);
    begin
      if TESTSYMINSet(SELECTSYS) then
       begin
         if TESTSYM(LBRACKET) then
          begin GETSYM;
            EXPLIST(FSYS+[RBRACKET]);
            CHECKSYM(RBRACKET,10)
          end
         else
            if TESTSYM(PERIOD) then
             begin GETSYM;
               CHECKSYM(IDENT,12)
             end
            else
               if TESTSYM(POINTER) then GETSYM;
         SELECTOR(FSYS)
       end
    end;

procedure FUNORVAR(FSYS:SYMSET);
    begin SELECTOR(FSYS+[LPAREN]);
      if TESTSYM(LPAREN) then
       begin GETSYM;
         EXPLIST(FSYS+[RPAREN]);
         CHECKSYM(RPAREN,8)
       end
    end;

procedure FACTOR(FSYS:SYMSET);
    begin
      TEST(FACBEGSYM,FSYS,107);
      if TESTSYMINSet(FACBEGSYM) then
       begin
         if TESTSYM(IDENT) then
          begin GETSYM; FUNORVAR(FSYS)
          end
         else
            if (TESTSYMINSet([INTNUM,REALNUM,STRING]))
            then GETSYM
            else
               if TESTSYM(NOTSYM) then
                begin GETSYM; FACTOR(FSYS)
                end
               else
                  if TESTSYM(LPAREN) then
                   begin GETSYM;
                     EXPRESSION(FSYS+[RPAREN]);
                     CHECKSYM( RPAREN,8)
                   end
                  else
```

```
                        if TESTSYM(LBRACKET) then
                         begin
                           GETSYM;
                           if not (TESTSYM(RBRACKET)) then
                              EXPLIST(FSYS+[RBRACKET]);
                           CHECKSYM(RBRACKET,10)
                         end
      end;
     TEST(FSYS,[],108)
   end;

procedure TERM(FSYS:SYMSET);
   begin FACTOR(FSYS+MULOPSYMS);
     if (TESTSYMINSet(MULOPSYMS)) then
      begin GETSYM; TERM(FSYS)
      end
   end;

procedure TERMS(FSYS:SYMSET);
   begin TERM(FSYS+[PLUS,MINUS,ORSYM]);
     if (TESTSYMINSet([PLUS,MINUS,ORSYM])) then
      begin GETSYM;
        TERMS(FSYS)
      end
   end;

procedure SIMEXP(FSYS:SYMSET);
   begin
     if (TESTSYMINSet([PLUS,MINUS])) then GETSYM;
     TERMS(FSYS)
   end;

procedure EXPRESSION;
   begin SIMEXP(FSYS+RELOPSYMS);
     if (TESTSYMINSet( RELOPSYMS )) then
      begin GETSYM;
        SIMEXP(FSYS)
      end
   end;

procedure STATEMENT(FSYS:SYMSET);
  forward;

procedure STATLIST(FSYS:SYMSET);
   begin
     STATEMENT(FSYS+[SEMICOLON]);
     if TESTSYM(SEMICOLON) then
      begin GETSYM;
        STATLIST(FSYS)
      end;
     TEST(FSYS,[],600)
   end;

procedure IFSTAT(FSYS:SYMSET);
   begin EXPRESSION(FSYS+[THENSYM]);
     CHECKSYM(THENSYM,16);
     STATEMENT(FSYS+[ELSESYM]);
     if TESTSYM(ELSESYM) then
      begin GETSYM; STATEMENT(FSYS)
      end
     else TEST(FSYS,[],601)
   end;
```

```
procedure WHILESTAT(FSYS:SYMSET);
   begin EXPRESSION(FSYS+[DOSYM]);
      CHECKSYM(DOSYM,17);
      STATEMENT(FSYS)
   end;

procedure REPEATSTAT(FSYS:SYMSET);
   begin STATLIST(FSYS+[UNTILSYM]);
      CHECKSYM(UNTILSYM,18);
      EXPRESSION(FSYS)
   end;

procedure OTHERSTAT(FSYS:SYMSET);
   begin SELECTOR(FSYS+[ASSIGN]);
      if TESTSYM(ASSIGN) then
       begin GETSYM; EXPRESSION(FSYS)
       end
      else
         if TESTSYM(LPAREN) then
          begin GETSYM;
             EXPLIST(FSYS+[RPAREN]);
             CHECKSYM(RPAREN,8)
          end
   end;

procedure STATEMENT;
   begin
      TEST(FSYS+[IDENT],FSYS,109);
      if TESTSYMINSet(STATBEGSYM+[IDENT]) then
       begin
         if TESTSYM(BEGINSYM) then
          begin GETSYM; STATLIST(FSYS+[ENDSYM]);
             CHECKSYM(ENDSYM,13)
          end
         else
            if TESTSYM(IFSYM) then
             begin GETSYM; IFSTAT(FSYS)
             end
            else
               if TESTSYM(WHILESYM) then
                begin GETSYM; WHILESTAT(FSYS)
                end
               else
                  if TESTSYM(REPEATSYM) then
                   begin GETSYM; REPEATSTAT(FSYS)
                   end
                  else
                     if TESTSYM(IDENT) then
                      begin GETSYM; OTHERSTAT(FSYS)
                      end
       end
   end;


procedure CONSTDEFPArt (FSYS:SYMSET);
   begin
      if TESTSYM(CONSTSYM) then
       begin GETSYM; CONSTDEFLIst(FSYS)
       end

   end;

procedure TYPEDEFPARt (FSYS:SYMSET);
   begin
      if TESTSYM(TYPESYM) then
       begin GETSYM; TYPEDEFLISt(FSYS)
       end
```

```
begin  (* Main Program *)

   DECLBEGSYM :=[CONSTSYM,VARSYM,TYPESYM,PROCSYM,FUNCSYM,FORWARDSYM];
   STATBEGSYM :=[BEGINSYM,IFSYM,WHILESYM,REPEATSYM];
   FACBEGSYM :=[LPAREN,NOTSYM,INTNUM,REALNUM,IDENT,STRING,LBRACKET];
   CONSTBEGSYm :=[PLUS,MINUS,INTNUM,REALNUM,STRING,IDENT];
   SIMPTYBEGSym :=[STRING,LPAREN,PLUS,MINUS,IDENT,INTNUM,REALNUM];
   SELECTSYS :=[POINTER,PERIOD,LBRACKET];
   TYPEBEGSYM :=[PLUS,MINUS,INTNUM,REALNUM,STRING,IDENT,LPAREN,POINTER
                    PACKEDSYM,ARRAYSYM,RECORDSYM,SETSYM];
   TYPDECL :=[RECORDSYM,ARRAYSYM,SETSYM];
   RELOPSYMS :=[EQSYM,NESYM,LTSYM,LESYM,GTSYM,GESYM,INSYM];
   MULOPSYMS:=[TIMES,SLASH,DIVSYM,MODSYM,ANDSYM];


   GETSYM;
   PROGRAMHEAd([SEMICOLON]+DECLBEGSYM+STATBEGSYM);
   BLOCK([PERIOD]+STATBEGSYM+DECLBEGSYM);
   IF NOT TESTSYM(PERIOD) THEN ERROR(21);
   if ERRCOUNT<>0 then
    begin
      WRITELN; WRITELN;
      WRITE (OUTPUT,ERRCOUNT);
      WRITE (OUTPUT,EMES);
      WRITE (TTY,ERRCOUNT);
      WRITE (TTY,EMES)
    end
   else
    begin WRITELN (OUTPUT,NOERRMESS);
      WRITELN (TTY,NOERRMESS)
    end
end.
```